A5: GREEDY ALGORITHMS

"The only two things you can truly depend upon are gravity and greed." — Jack Palance

Course: CS 5006

Summer 2018

Due: 15 June 2018, 5pm

OBJECTIVES

After you complete this assignment, you will be comfortable with:

• Greedy algorithms

RELEVANT READING

• Kleinberg and Tardos, Chapter 4

AND NOW ON TO THE FUN STUFF.

Problem 1: Web Site Analytics (10 points)

In this problem, we're concerned with a particular kind of web site analytics. Sometimes, as we're attempting to optimize the design of a website, we want to understand how people use the website. Websites are frequently instrumented with 'analytic hooks' that allow us to track, in a given session, what interactions a site visitor ("user") makes on the site, such as clicking a particular link, viewing a certain product, etc. I want to use this data to validate hypotheses I have about what a user is doing on the website.

For example, I might hypothesize this sequence of actions:

Seq1: View the About page, Fill out the Survey, Create an account

Given a sequence of actions a user takes on the website, I want an efficient algorithm that will determine if that sequence of actions happens in that order.

A sample sequence of actions a user takes on the website:

Actions: View the home page, View the Contacts page, View the About page, View a Testimonial, Fill out a Survey, View a Blog post, Create an Account

I want my algorithm to return TRUE that Seq1 exists in the Actions sequence.

(a) (2 points) Provide a couple of sentences on how this problem is similar or different to problems we've already seen.

This problem is similar to problem's we've already seen because we have a set number of nodes or items we are considering, the actions that the user can take on the webpage. So the sequence of user actions is kinda like a directed graph since the user is doing those actions in that order. But it's different in that while in other problems, we are trying to connect the nodes in some way, optimize the path to get to a node, or, display the graph in a different way, here we aren't trying to change the path, we are trying to see whether our prediction matches the path in terms of order.

(b) (4 points) Give an algorithm to solve the problem in time $O(m + n)$, where m is the length of a target sequence S_T, and n is the length of an actual sequence, S_A. The algorithm returns TRUE if S_T exists in S_A.

```
CheckSequence(SequencePrediction, Sequence){ assume sequence Prediction and sequence are queues
        lengthPrediction  = SequencePrediction.length;
        correctCount = 0;
        While Sequence != null{
                If(SequencePrediction[0] != Sequence[0] && sequence.length> 1){
                        Pop(Sequence[0]);}
                Else If(SequencePrediction[0] == Sequence[0] && sequence.length> 1) {
                        Pop(Sequence[0]);}
                        Pop(SequencePrediction[0]);
                        correctCount += 1;}
                Else{
                        If (SequencePrediction[0] == Sequence[0]){
                                Pop(sequence[0]);
                                correctCount += 1}
                        Else{
                                Pop(sequence[0]);}
        }

        If(correctCount == lengthPrediction){
                Return true;}
        Else{
                Return false;
```

(c) (4 points) Prove your algorithm correct.

So given that the goal is to see whether the actions happened in the order the user predicted, an optimal solution would return an accurate assessment of whether the sequence occurred as the prediction stated it would and didn't go through either the prediction or sequence array more than once. So let O be the optimal solution, which in the case of the example to the problem, would be true. My algorithm also returns true from the example in the problem.

While the optimal algorithm doesn't go through my prediction or sequence array more than once, my algorithm does the same. For example, say, if we are making the sequences letters instead of actions, the predictive sequence was A, B, C, and the actual sequence was: D,F,G, B, Y, W, A, C, Q. The optimal solution would look at D and A, and not find a match, F, and A, and not find a match, keep going until it found an A in the sequence. My algorithm does the same; it goes through each element in the sequence array only once because as soon as it looks at the first element in the array, it removes it. So that portion will be O(n), the same run time as the optimal, since you have worse case, you would go through all the elements of the sequence array when looking for a match.

And in the optimal algorithm, once it makes A, it looks for B in the remainder of the sequence, because we are checking order, not just the fact it's there. My algorithm does the same. Additionally, it may not go through the sequence prediction array in < O(m) time because in a situation where the actual sequence isn't in the same order and the first element of the prediction is found later on, by the time you get through the sequence, you won't get through the rest of the prediction array . In those cases, if the optimal if O(m) when going through the prediction array, my algorithm would be faster.. Since my algorithm is better or equal to the running time of the optimal, my algorithm is the optimal

Problem 2: Amazon Product Development (4 points)

As a new product manager at Amazon, I can't help but think about ways that Amazon could make more money. Specifically, I'm aware that sometimes customers get partially-filled, almost-empty boxes, and the customer might not be so concerned about shipping time that maybe we can provide an alternative shipping option that costs us less, doesn't unduly inconvenience the customer, and hopefully has a positive environmental impact by shipping fewer boxes in general.

Here's what I propose: as a customer, when shopping on Amazon, I start purchasing things. Rather than (essentially) every purchase being put into a box and shipped to me, the items go in a queue. When I have purchased enough items to fill a box, ship those items. Because of the way shipping and transportation works, the definition of a "filled box" is based on weight, not volume filled. That means, when I've purchased a set of items that meet a certain weight, ship them out.

For the purposes of simplifying this problem, assume that all shipments go in an "optimal" sized box to fit the items that meet the shipping weight.

As a product manager, I want to prove that this approach minimizes the number of boxes we will use to ship products to a customer. Prove that this greedy strategy does actually minimize the number of boxes.

Hint: Follow the analysis done by the Interval Scheduling Problem.

Assume that the items in a box are compatible if their total weight combined is close to or equal to the weight we are trying to hit. Assume also that we are continually sorting the queue as we add new items based on the items weight with the lightest being at the front of the queue. Let i_1…,i_k represent the items in the box for our algorithm in the order they were put in. Let O be an optimal algorithm and j_1….,j_m represent the items in that box(same items). Assume the items in the box for O are compatible. Since our goal is to fill up the box with as many items as possible before reaching the weight limit, we want to first item in the box to be the lightest item there is. So, $w(i_1) \leq w(j_i)$, since the optimal will put in the lightest item it has in its queue and our algorithm does the same.

Let r be some element in the queue. If r = 1, we've established that our algorithm with have an item or equal or lighter weight than O. We assume that $w(i_{r-1}) \leq w(j_{r-1})$. For our algorithm to be worse than the optimal, the next item the algorithm chooses for the box must be greater than the weight of r^{th} item for the optimal. That's not possible because our algorithm has the chance to choose $w(j_r)$ instead of a heavier items for its r^{th} element. Since it always chooses the lighter item, our algorithm would chose $w(j_r)$ for $w(i_r)$. Therefore, $w(i_r) \leq w(j_r)$. Since our algorithm will chose an item that is either lighter or the same weight as the item the optimal chooses at that same step, our algorithm is optimal. And since our algorithm is optimal is filling boxes, it will minimize the number of boxes used.

Problem 3: Cell tower placement (4 points)

Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible, with 1-2 sentences describing your approach.

```
CellPlacement(House){
    Let section = 0;
    Choose arbitrary house;
    Create Array[amount of houses];
    Array[section] = arbitrary house;
    Remove arbitrary house from House;
    Sort houses by distance from arbitrary house;
    While House != null{
        If House[0] is within 4 of arbitrary house{
            Link house to arbitrary house list;
            Remove house from House;}
        If house is not within 4 of arbitrary house{
            Drop cell tower near arbitrary house;
            Section +=1;
            Array[section] = house;
            Arbitrary house = array[section];
    }
}
```

So, I thought the easiest way to do it would be to divide the houses into sections and to put cell towers within each of those sections to find the minimum amount of cell towers to use. So I split the houses based on their distance from a head/arbitrary house, dropped a cell tower near the house since all the houses in that section were within 4 miles of that house, and changed the head house once we reached houses that were too far away and continued with the rest of the houses.

Problem 4: Road Trip! (6 points)

You and your friends are planning a road trip to a bunch of National Parks this summer. Good CS students that you are, you model the route as a directed graph, where the nodes represent the destinations and the edges the path between them.

It happens that over the summer, some roads are slow for different reasons: sometimes there's road work scheduled to be done, or events at the park which result in a predictable slow down. However, there's a web service that provides a very good estimate of the time it takes to travel a path on a particular day.

That is, given an edge $e(v, w)$ connecting two parks v and w, and a datetime t that indicates the starting date and time from location v the service returns a value $r(t)$ that is the predicted arrival time at w.

The service guarantees that $r(t) > t$ for all edges e and times t, which really just means all edges take a non-negative time to travel on them.

Now that you have this service, use it to find the fastest path through the graph from the starting point to the destination. Assume that you start at time 0, and all predictions provided by the service are accurate. Provide a polynomial-time algorithm that does this, assuming that a single call to the service is a time cost of 1. In addition to the pseudocode, provide a 1-2 sentence explanation of your approach.

```
FastestRoute(RoadMap){
    Let weight represent r(t) – t;
    Create array of weights;
    Sort array of weights
    Create nodetraveltimearray[amount of nodes];
    Zero nodetraveltimearray;
    Assign each location an index in the array;
    For(i=0; i<amount of edges; i++){
        If (nodetraveltimearray[locations in array] == 0){
            nodetraveltimearray[location in array] = weight; - do this for each location associated with
the weight.
        Else{
            If(nodetraveltimearray[location in array] > weight){
                If (nodetraveltimearray[location in array]->next < weight){
                    Weight->next = nodetraveltimearray[location in array]->next;
                weight->prev = nodetraveltimearray[location in array];
                    nodetraveltimearray[location in array]->next = weight;}

                Else{
                    Go through list until above requirement hits;}

    For (i=0; i<amount of nodes-1; i++){
        As long as this doesn't split RoadMaps{
            Node = node->next;}
        }
    }
}
```

So I thought the best way to do this was to get an array, where each array index represents a node, and within each index, there is a list of all the weights from heaviest to lights. My thinking was outside of the starting location, that if you keep going through the list, popping the top node of each node list until you have the edges that make sure the node stays attached to the larger graph, you guarantee each, and you have them shortest route through the parks.

Problem 5:Ternary Huffman(6 points)

Generalize the Huffman algorithm to support ternary codes rather than binary (that is, rather than producing a code using 1's and O's, use 0, 1 and 2).

Prove that the algorithm produces optimal ternary codes.

```
To construct a prefix code for an alphabet S, with given frequencies:
        If S has three letter then
                Encode one letter using 0, one letter using 1, and one the last letter using 2
        Else
                Let x, y, and z be the three lowest-frequency letters
                Form a new alphabet S’ by deleting x, y, and x and replacing
                        Them with a new letter q of frequency $f_x + f_y + f_z$
                Recursively construct a prefix code $g$’ for S’, with tree T’
                Define a prefix code for S as follows:
                        Start with T’
                Take the leaf labeled $g$’ and add three children below it named
                        x, y, and z
        End if
```

Proof attempt:

Suppose by way of contradiction the algorithm doesn’t produce an optimal tree. So there is a optimal ternary tree Q that represents x, y and z as siblings (assuming these pop up with the lowest frequency). So the amount of bits needed to encode Q is less than the amount of bits needed to encode our tree. So delete x, y, z from Q and name their former parent w. So, the average amount of bits needed to encode each letter in Q’ is equal to the amount of bits needed to encode each letter in Q minus the frequency of w or $ABL(Q') = ABL(Q) - f_w$. (using 4.32 from book). If we were to do the same with
our tree, we’d get end up with the same result, a tree where the average amount of bits to encode each letter is $ABL(T') = ABL(T) - f_w$. If our tree isn’t optimal, than the amount of bits to encode our T’ has to be bigger than the amount needed to encode Q’. Then T’ isn’t optimal, despite it needing to be optimal to define the prefix for S’. This is a contradiction, thus it our algorithm is optimal.

Problem 6: Variable vs Fixed Length Encoding (4 points)

Suppose you have a file that contains a bunch of 8-bit characters. All 256 characters are about equally common: the max character frequency is less than twice the minimum character frequency.

Prove that in this case, Huffman encoding is no more efficient than using an ordinary 8-bit fixed-length code (such as ASCII).

So if the characters are equally common, as you build the tree, the depth of the tree will be really high, since the frequency is pretty equal. So the weighted average will be pretty high.

As you start building the tree, the frequency differences are so low that you keep having low frequency values you need to account for as you are building the tree. So you keep having to branch out before you get to a point that you can start adding parents together or parents to a node since the algorithm requires you to combine the two lowest amounts into a new node each time. Since there are 256 characters, worse case scenario is that you end up with 128 parents. The same issue on the lower level continues happening; 128 becomes, 64, becomes 32, become 16, becomes 8, become 4, become 2, become 1. So with a possibility of 8 as a worse cause depth, the weighted average depth will be so high that the amount of bits needed to encode each character could be end up being 8 bits. Which makes it just as good as ASCII.

Question	Points	Score
Web Site Analytics	10	
Amazon Product Development	4	
Cell tower placement	4	
Road Trip!	6	
Ternary Huffman	6	
Variable vs Fixed Length Encoding	4	
Total:	34	

SUBMISSION DETAILS

Things to submit:

- Submit your assignment in your Github repo.
  - The written parts of this assignment as a .pdf named "CS5006_[lastname]_A4.pdf". For example, my file would be named "CS5006_Slaughter_A4.pdf". (There should be no brackets around your name).
  - Make sure your name is in the document as well (e.g., written on the top of the first page).
  - Make sure your assignment is in the A4 folder in your Github repo.

HELPFUL HINTS

- Ask clarification questions on Piazza.
- Remember, your write-up should convince graders and instructors that you are providing your own work and should showcase your understanding.
- Use the resources page on the course website for supplemental materials.
- In general, problems will be graded both on whether you are taking the right approach and whether you got the right answer. So, show your work and explain your thinking.